About This Book

This document describes the Enhanced Program Maintenance Utility (NMAKE32). This is a new utility that provides a superset of the features found in the "legacy" NMAKE utility. Since there are some behavioral differences between the new and old versions of the tool, they are being provided separately so that users can select the version that works best for them.

Enhanced Program Maintenance Utility (NMAKE32)

Purpose

The Enhanced Program Maintenance Utility (NMAKE32) automates the process of updating project files and can be used to make backups, configure data files, and run programs when data files are modified.

Description

The Enhanced Program Maintenance Utility (NMAKE32) compares the modification dates for one set of files (the target files) with those of another set of files (the dependent files). If any dependent files have changed more recently than the target files, NMAKE32 executes a series of commands to bring the targets up-to-date.

Program Maintenance Utility Program (NMAKE32)

The purpose of the NMAKE32 tool is to automate the process of updating project files. NMAKE32 compares the modification dates for one set of files (the target files) with those of another set of files (the dependent files). If any dependent files have changed more recently than the target files, NMAKE32 executes a series of commands to bring the targets up-to-date.

Why Use NMAKE32?

Large projects tend to have many source files. Often, only a few source files need to be compiled when a change is made. NMAKE32 allows a user to set up a special file called a "description" file (or "makefile") that tells NMAKE32:

- Which files depend on others
- Which commands, such as compile and link commands, need to be carried out to bring programs up to date.

This use of NMAKE32 is only one example of its power. By building suitable description files, NMAKE32 can be used to:

- Make backups
- Configure data files
- Run programs when data files are modified.

Upgrading from prior versions of NMAKE

Relative to older versions of NMAKE, NMAKE32 provides a richer set of functions including:

Transformation macros	Transformation macros manipulate strings and can be used for pattern matching and for conversions from upper to lower case

	and vice versa.
Built-in commands	Built-in commands are commands processed internally by NMAKE32 to perform basic operations such as changing directories or setting environment variables.
Built-in functions	Built-in functions are provided that can be used in conditional constructs to test such things as the return code from the last command executed.
New and improved directives	Directives can now be used at run-time as part of a command block. Two new directives, !elseif and !tryinclude have been added.
New macro assignments	Macro assignments can now use =+ and += to append a string to either the beginning or the end of an existing definition.

Be sure to read Migration Considerations for a few differences between NMAKE32 and previous versions of NMAKE.

Migration Considerations

NMAKE32 is fully compatible with previous versions of NMAKE with the following exceptions:

- When the -n option is specified, only commands prefixed with & or $(MAKE) are executed.
- Parentheses must be used to group expressions in directives that contain multiple expressions.
- Control characters used in filenames and pathnames in a makefile must be preceded by a caret (^) character so that they are not evaluated by NMAKE32. This is especially important in pathnames where an ending backslash character could erroneously be treated as the line concatenation operation by NMAKE32 instead of the literal backslash character intended by the makefile writer.
- A pseudo-target without a dependent must contain a command block.

Command-Line Syntax

NMAKE32 accepts certain flags on the command line, each of which may be specified in upper or lowercase (or any combination). Flags may be listed together following a single "-" or "/" sign. For example, the following invocations are equivalent:

```
nmake32 -N /d -P

nmake32 /n -dP

nmake32 /ndp
```

Some flags require an additional value. In every case, the value must be specified as a separate argument at invocation. If more than one flag that requires a value is given, the values are assigned in the order that the flags were specified. For example, the following invocations are equivalent:

```
nmake32 -FBD mymake.fil mybuilt.ins

nmake32 -D -F mymake.fil -B mybuild.ins
```

Following is the general syntax for NMAKE32:

```
nmake32  [flags]  [targets]  [var=val]  [@responsefile]
```

Command-Line Options

The flags can be one or more of the following options:

-?, -h, -help	Display the help information and exit.
-a	Rebuild all targets regardless of the date or timestamp of the dependent files.
-b *builtins_file*	Use the description file specified by *builtins_file* instead of the default description file name of BUILTINS.MAK.
-c	Suppress the display of the NMAKE32 sign-on banner, nonfatal error messages, and warning messages.
-d	Display a trace of the execution as it checks each target against its dependents and prints the dates. NMAKE32 also prints a macro substitution trace.
-e	The values of environment variables will override the values in the makefile.
-f *filename*	Use the description file specified by *filename* instead of the default description file name of MAKEFILE. If the specified description file is left blank, then the description file is read from standard input.
-i	Do not terminate execution when a command returns a non-zero return code. Continue to process all targets.
-k	Do not terminate execution when a command returns a non-zero return code. Continue to process targets which do not depend on the failed target.
-l	Do not combine contiguous spaces into a single space.
-n	Instead of directly invoking the commands, NMAKE32 will only display the commands that would have been executed based on the evaluation of conditionals. The $(MAKE) command and commands prefaced with & are always executed. All commands are displayed, even those prefaced with @. It is important to prefix any command that must be executed when using this option with &. For instance, if a change directory (CD) command must be performed for NMAKE32 to locate a file, this CD command must be prefixed with an & so that the appropriate file is found when using the -n option.
-o[+] *batch_file*	Instead of directly invoking the commands, NMAKE32 creates the specified custom batch_file listing the commands. The flag -o will overwrite the specified batch_file and the flag -o+ will append to the specified batch_file. If the specified file cannot be opened, NMAKE32 will write the commands to the standard output device. The custom batch file will contain only those commands which are needed, no extraneous messages will be written.
-p	Prints out the contents of the internal data structures: macros, rules (targets and dependents), and commands.
-q	Queries the target and determines if it is up to date. Exits with a "0" return code if the target(s) are up to date, or returns the number of out-of-date targets if the target(s) are not up to date. Also, this sets the -n and -s flags.
-r	Reject default values mode. Does not read the default TOOLS.INI or BUILTINS.MAK files. Does not use the built-in extensions for inference rule processing (.SUFFIXES). Does not define default macros AS (masm), CC (icc), COBOL (cobol), FOR (fl), PASCAL (pl), or RC (rc). Does not define default inference rules.
-s	Do not display the commands as they are executed. This has the same meaning as the .SILENT: target in the makefile.
-t	Touch all out of date targets with the current date and time without rebuilding them. If a target does not exist, it is created with the current date and time and a length of 0.
-v	Macro names are case-insensitive as opposed to case-sensitive. Macros defined from the environment will be defined in uppercase.

-x *errfile*	Redirects error messages (stderr) to the specified file, errfile.
-\	Do not place a space between continued lines. The two lines are to be continued exactly as they appear in the description file after the backslash is removed.
-=	Select whether or not to build when target/dependent timestamps are equal. The default is to build when the timestamps are equal.
-nologo	Suppresses the NMAKE32 sign-on banner without suppressing other messages.

Other command-line options are:

targets	A list of one or more targets to be built. Because OS/2 is case-insensitive, *targets* can be specified in upper or lower case. If a target is not specified, NMAKE32 will attempt to rebuild the first target specified in the description file.
var=*val*	Creates the variable var and assigns it the value *val*. If there are any spaces in this option, it must be enclosed in double quotes. This definition overrides all other definitions of this variable.
@responsefile	The file, responsefile, contains additional command-line input to NMAKE32.

Using NMAKE32 Response Files

Input to NMAKE32 can be split between the command line and a response file. The name of a response file (preceded by @), can be used as a command-line option. A response file can be used for:

- Complex and long commands
- Strings of command-line arguments, such as macro definitions, that exceed the limit for command-line length.

Command File Syntax

```
nmake32 @responsefile
```

NMAKE32 treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines if each line is terminated with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

The following is an example of a response file called UPDATE:

```
 /s "program\
= flash" SORT.EXE SEARCH.EXE
```

To use this response file type the following command:

```
nmake32 @update
```

This response file runs NMAKE32 using:

- The /s option
- The macro definition "program = flash"
- The targets specified as SORT.EXE and SEARCH.EXE
- The description file MAKEFILE by default

Note that the backslash allows the macro definition to span multiple lines.

The NMAKE32 Environment

NMAKE32 automatically accesses all environment variables, converting each into a macro of the same name. Following is the evaluation precedence of macros:

Priority	Definition
1 (Highest)	Macros from the command-line
2	Macros from the description file and its include files.
3	Environment variables.
4	Macros from TOOLS.INI or BUILTINS.MAK
5 (Lowest)	Predefined macros (such as MAKEDIR or TIMESTAMP)

Note that macro definitions found at a higher priority replace macro definitions at a lower priority (for example, macros from the description file replace macros from the built-ins file).

The -e flag causes macros defined from environment variables to overwrite previously defined macros (switches the processing of numbers 2 and 3).

The MAKEFLAGS environment variable is generated by NMAKE32 to contain all input flags specified on the NMAKE32 command line, except for the -b, -f, -x, -o and -p flags (and their corresponding values).

If it is not already in the environment, NMAKE32 creates the MAKEFLAGS variable and sets it according to the current flags, and passes it on to nested NMAKE32 invocations as they are called. Therefore, MAKEFLAGS always contains the current input flags.

Each time NMAKE32 is executed, it reads the MAKEFLAGS variable from the environment. If it exists and is non-null, its contents are processed before the NMAKE32 command line is processed.

The TOOLS.INI File

NMAKE32 can use a file named TOOLS.INI for initialization information. NMAKE32 looks for the TOOLS.INI file in the current directory, and then in the directory specified by the INIT environment variable. If NMAKE32 finds a TOOLS.INI file, it looks for the following section name:

```
[nmake]
```

Note: This section name is case-insensitive. If a TOOLS.INI file is to be processed, it must contain an "[nmake]" section, even if the section is empty. NMAKE32 will generate a warning if this section name is missing, and will attempt to process the BUILTINS.MAK file.

The only valid information the user can place in this section are macros and inference rules using the same format specified for description files. However, if a description file and TOOLS.INI both contain a definition for the same macro or inference rule with the same extensions, the definition in the description file overrides the definition in TOOLS.INI. The TOOLS.INI file can be used to customize default options for NMAKE32. For example, if an inference rule is needed for several description files, instead of modifying all the description files, the rule can be added to the TOOLS.INI file.

Example:

Given the following TOOLS.INI excerpt:

```
    [nmake]
    CFLAGS=
    .c.obj:
        $(CC) -c $(CFLAGS) $*.c
```

These lines:

- Define the CFLAGS macro as a null string
- Redefine the predefined inference rule to build .obj files from .c source files using the CFLAGS macro.

The BUILTINS.MAK File

In addition to the TOOLS.INI file, NMAKE32 supports the use of an optional file named BUILTINS.MAK. NMAKE32 will process either the TOOLS.INI or the BUILTINS.MAK file. If a TOOLS.INI file is not found, or the nmake section does not exist, NMAKE32 automatically searches for the BUILTINS.MAK file in the current directory or the file as specified by the -b flag. When an initialization file is processed, it is included as part of the makefile currently being executed. This feature allows the user to supplement the set of predefined macros that NMAKE32 supports and to maintain an additional set of inference rules.

Like the TOOLS.INI file, the BUILTINS.MAK file can be used to customize default options for NMAKE32. For example, if an inference rule is needed for several description files, instead of modifying all the description files, the rule can be added to the BUILTINS.MAK file. Unlike the TOOLS.INI file, an "[nmake]" section is not expected. All lines from the file are expected to be valid description file statements.

Example:

The lines in the BUILTINS.MAK file below do the following:

- Define the CFLAGS macro as a null string
- Redefine the predefined inference rule to build .obj files from .c source files using the CFLAGS macro

```
CFLAGS=
.c.obj:
    $(CC) -c $(CFLAGS) $*.
```

Description Files

The file that contains the set of customized rules for NMAKE32 is a special file referred to as a description file. The default description file that NMAKE32 will search for is named **makefile**. The /f option is used to provide a description file name when the user does not want to use the default description file. NMAKE32 will return an error if a description file is not provided. A description file consists of the following elements:

- Description blocks
- Macro definitions
- Inference rules
- Directives

The following syntax rules apply to description files:

- A target/dependency statement must begin in column one.
- A command line must begin with a space; it cannot start in column 1.
- A backslash(\) at the end of a line tells NMAKE32 that the next line is a continuation of the current line.
- An unescaped pound sign (#) anywhere on a line in a description file indicates that the remainder of the line is a comment. Since the lines in an in-line file are processed by the command shell, specifying comments is specific to the shell in use. In OS/2, CMD.EXE uses **rem**.

NMAKE32 uses the target/dependency rules in the description file to determine if a target needs to be updated. All dependents are recursively checked for being out of date with respect to their dependents. Missing targets are considered to be out of date. If a target is determined to be out of date, the command associated with the target/dependency statement is executed. See the following example:

```
program.exe:    program.obj  abcd.obj
    ilink program abcd;

program.obj:    program.c   xxx.h
    icc program.c

abcd.obj:       abcd.c      xxx.h
    icc abcd.c
```

The above example description file says that program.exe depends on two files, program.obj and abcd.obj, and that they in turn depend on their corresponding files, program.c, abcd.c and a common include file, xxx.h.

When NMAKE32 is executed with the above description file, it would determine that program.exe was the main target and that abcd.obj and program.obj are the files that it depends on. NMAKE32 would then recursively determine that program.obj depends on program.c and xxx.h, and that abcd.obj depends on abcd.c and xxx.h. Thus, NMAKE32 constructs a graph of file dependencies. Starting from each node of the graph, NMAKE32 compares the modification date of each node (file) to its immediate descendants. If a file is older than one or more of its descendants, then the command associated with the file is executed to bring the file up to date.

When the above description file is processed by NMAKE32, assuming that each .obj file is older than the corresponding .c file, the following commands would be executed:

```
icc -c program.c
icc -c abcd.c
ilink program abcd;
```

These are all the necessary commands to make program.exe from program.c, abcd.c, and xxx.h. If any of these commands encountered an error, NMAKE32 would immediately terminate and report which command failed.

If the file abcd.c was modified, then executing NMAKE32 again would only execute the commands necessary to bring program.exe up to date. The commands executed now would be:

```
icc -c abcd.c
ilink program abcd;
```

If the file program.exe was missing, executing NMAKE32 would execute the following command:

```
ilink program abcd;
```

If the file xxx.h was modified, executing NMAKE32 would cause all the commands to be issued again because abcd.obj and program.obj depend on it.

Case Sensitivity

NMAKE32 statements within a description file are case insensitive, except macro names. Macro names can be changed to be case insensitive by using the -V option. For example, the !IF, !if, and !If are all correct forms of the if directive.

Read-Time vs. Run-Time

Read-time processing occurs during the reading of the description file, in a similar manner to the C language's preprocessor stage, and essentially determines which parts of the description file are used. At read-time all of the statements beginning in column 1 are processed. During read-time processing, the graph of all targets and their dependents is built, macros are initially defined, and directives are processed. Run-time processing begins after read-time processing has completed, and is the stage during which NMAKE32 checks file dates/times, and invokes any required commands. At run-time the statements that are not in column 1 are processed.

Every directive may be used during either read-time or run-time, with different effects.

Description File Elements

NMAKE32 uses a description file to determine a series of actions that must be executed to update a target. In its simplest form, a description file indicates which commands need to be executed when a dependent file is updated.

Comments

The pound sign (#) tells NMAKE32 that what follows is a comment and not a description file statement to be acted on. NMAKE32 ignores blank lines in description files. The pound sign cannot be used in in-line files to indicate comments (see In-Line Files).

A comment can be on a line by itself or on the same line with a description file statement that will be executed. When a comment is on a line by itself, the pound sign can be anywhere on the line. When the comment is on the same line as an executable description file statement, the pound sign follows the description file statement and everything that follows the pound sign on that line is considered to be a comment and is therefore ignored by NMAKE32.

Comment lines cannot be continued with the backslash character.

The pound sign can be used as a literal character if it is preceded by a caret (^). See Escape Characters for more information.

Example:

```
#  The following description file shows examples of comments:

# This is a comment line.

program.exe: program.obj abcd.obj # This is a comment.
       ilink program abcd;

# These three lines, 1,
#  2, and
#  3, are comment lines.

program.obj:     program.c  xxx.h
       icc -c program.c

abcd.obj:        abcd.c     xxx.h
       icc -c abcd.c
```

Description Blocks

A description file consists of description blocks that define the dependent relationship between files. The description block specifies the creation and/or update process for each file. A description block has the following format:

```
targets...  :  dependents...
   command
   .
   .

targets...  :  dependents...
   command
   .
   .
```

Example:

```
# Program build description block

program.obj : program.c program.h
```

```
    icc /c program.c /Foprogram.obj

# End of program build description block
```

Targets and Dependencies

Targets are the files that will be created or updated in the description file. Targets are file names that start in column one of a description file and are followed by a colon (:). Dependents are the filenames that follow after the colon. These are the files that are examined to determine if the target should be updated. Specification of the target(s) and dependent(s) will be frequently referenced as the target/dependency statement. The target/dependency statement must start in the first column of the description file by definition. The target/dependency statement has the following format:

```
targets...  :  [{path}]dependents...
```

or

```
{path1;path2;...}targets  :  {path1;path2;...}dependents
```

Example:

```
program.obj : program.c program.h
```

Note: Target/Dependency statements may not contain an equal sign (=). Only macro definitions may contain an equal sign.

The optional curly brace paths preceding the filenames are alternative paths that can be used if the file is not located in the current directory. When specifying multiple paths they must be separated by semicolons (;). Whenever one of the dependents has a timestamp equal to or newer (depending on the use of the -= option) than the target, the target will be updated. All dependents are also checked for being out of date with respect to their dependents. Missing files are always considered to be out of date. The target build process can be specified in a command block following the target/dependency statement. If there is no command block in the description block, then the build process will be inferred from either rules predefined in NMAKE32, or rules that have been given in the description file, the TOOLS.INI file, the BUILTINS.MAK file, or the file specified by the -b option on the command line (see Inference Rules).

NMAKE32 builds the first target that is specified in the description file if it is out of date. Often a pseudotarget will be used to list all the targets that the developer wants to build from the description file.

Example:

```
all : hello.obj bye.obj

hello.obj : hello.c hello.h
   icc /fohello.obj /c hello.c

target1.obj : target1.c
   icc /fotarget1.obj /c target1.c

bye.obj : bye.c bye.h
   icc /fobye.obj /c bye.c

target2.obj : target2.c
   icc /fotarget2.obj /c target2.c
```

In the preceding example, of the four targets listed (five, including the pseudotarget **all**), only hello.obj and bye.obj will be checked for updating. The first target NMAKE32 encounters is **all**, and it has two dependents, hello.obj and bye.obj.

If we deleted the target/dependency statement that defines **all** and its dependents, the only target checked for updating would be hello.obj, because it would be the first target in the description file.

Targets can also be specified during NMAKE32 invocation from the command line. When targets are specified during invocation, they become the targets to be updated, those targets and their dependents will be checked for updating. The command line targets supersede targets defined in the description file. If a command line target does not correspond to a target in the description file, NMAKE32 will attempt to infer a command using either a default inference rule, or an inference rule defined in the makefile.

A target specified in the description file without dependencies is always out of date. When specified for updating, it's command block will always execute.

Commands

A command block is the set of commands that must be executed in order to build a target. These commands usually consist of compile statements or file processing commands (such as executing tool commands, file manipulation, directory changes, or system commands). All lines following a target/dependency statement that start with a space are executed as commands. A command can also be specified on the target/dependency statement line, if preceded by a semi-colon (;). The format is the following:

```
targets...  :  dependents; command
```

or

```
{paths;...}targets  :  {paths;...} dependents; commands
```

Example:

```
{\program\objs; \objects}program.obj : {\source} program.c \
program.h;icc /c program.c /Foprogram.obj
   @echo Build Finished
```

The first line following the target/dependency statement that does not begin with a blank or a comment indicator (#) is taken as the beginning of a new target/dependency or macro.

If a command does not fit on one line, it can be continued on the following line by using the line continuation character (\) at the end of the line. The remainder of the line can be typed on the following line. Whenever a backslash character is found, it is deleted and the following line appended to the current line.

```
always:
    @echo This command line will always execute and will \
                         continue on the next line!

#  And this line is just a comment line
```

Dependent Specific Rules

There is a method of choosing a command block to be executed based on which dependents cause the target to be updated. If an object library contains both compiled files and assembled files, you might want to generate the objects and update the library in different ways depending on which file types were updated. NMAKE32 supports this with dependent specific rules. These rules are similar to regular rules, but are identified by using two colons (::) between the target and the dependents, and the target is usually specified in more than one description block. The following description blocks are permissible:

```
X :: a
   command block

X :: b
   command block
```

The following example shows how dependent specific rules can be utilized.

```
object.lib :: a.asm
 $(AS) a.asm;
 $(LIB) object -+a.obj;

object.lib :: b.c
 $(CC) $(CFLAGS) b.c
 $(LIB) object -+b.obj;
```

The two description blocks both update the library named object.lib, however, the first command block is only executed if a.asm is newer than object.lib, and the second command block is only executed if b.c is newer than object.lib.

Whenever there is a target that appears in more than one description block, and the target/dependency separator is a single colon (:), the two description blocks are merged by concatenating the command blocks and the dependents.

Wildcards

In some cases, it is not desirable to have to specify all the filenames that are required to build a target. OS/2 wildcards ('*' and '?') are supported on the dependency file specification.

Wildcards cannot be used in a dependency list to reference files that do not exist at the time of execution of NMAKE32. Files that will be generated (dependents from one target/dependency statement that are targets in another target/dependency statement) during NMAKE32 execution cannot be referenced using wildcards.

Wildcards cannot be used on targets or for path specifications.

Explicit Command Processing

Explicit commands are contained in the command block that follows a target/dependency statement. It is the block of commands that will be executed if the target needs updating.

Example:

```
targ.obj :  program.c            #  Target / dependency statement
  icc /c /Fotarg.obj program.c
```

Whenever there is an explicit command associated with a target/dependency statement, inference rule processing is not attempted.

Paths may be specified for targets and dependents. A target file in the specified target directory will be built or updated if the dependent is found in the specified directory.

When paths are specified in curly braces on file specifications, the default is to use the local directory. If a file is not found that matches the file specification in the local directory, then the path in curly braces (or paths, separated by semicolons) is searched in the order given.

A file can be both a target and a dependent in a description file. If the file is specified with curly brace paths in any instance, all instances of that file must contain the same curly brace paths. If the file is not specified the same way in all instances, it will be treated as a different file specification.

When specifying multiple search paths in curly braces on targets and dependents, the paths are specified in the following manner:

```
{p1;p2;p3} targ.obj : {p4;p5;p6} dep.c   # Multiple paths in curly braces
    $(CC) -c -Fo$@ $<                    # Explicit rule for target/dependency
```

File specifications can be combined with relative path specifiers to modify the local directory or to modify the paths in curly braces. The relative path specifier is included on the file specification. The relative path specification works in OS/2 in the following manner:

```
cd e:\os2\system\test     # executing this from the e: drive
                          # would locate you in the specified
                          # directory

dir ..\..\filename.txt    # This file specification includes a
                          # relative path specification.  It
                          # will look for filename.txt in the os2
                          # directory
```

.PATH macro processing is used to search for unlocatable files after any applicable curly brace paths have been processed. .PATH searches are applied if the unlocatable files match the .PATH macro extensions and do not have a path or relative path included in their file specification. The .PATH macro processing method of searching for files works only with target/dependency statements that have an explicit rule.

.PATH processing functions similar to explicit curly brace processing in that it will use the first file it finds, starting with the .PATH processing functions similar to explicit curly brace processing in that it will use the first file it finds, starting with the local directory and then searching the paths specified by .PATH.

For more information on the .PATH macro, see Predefined Macros.

Inference Rules

Inference rules allow you to define rules that can be used for more than one file. When NMAKE32 encounters a target/dependency statement with no commands, it looks for an inference rule that specifies how to create the targets based on the defined .SUFFIXES extensions, the defined inference rules, and the target/dependent base file names.

In order for an inference rule to be applied, three conditions must be met:

- The extensions of the target file and the dependent file must be defined in the .SUFFIXES: target
- There must be a valid inference rule with correct path specification defined.
- The target and dependent must have the same base file name.

A valid inference rule, with local directory path specification, starts in the first column of the description file, and consists of the dependent extension, followed by the target extension and a colon. The command block starts on the following line, and must be preceded by at least one space.

```
.extension1.extension2:
    [command..]
```

Given a target/dependent statement, with the same base file name, and valid file extensions in the .SUFFIXES: target, NMAKE32 will search for a defined inference rule, and execute the rule if available. If the target and all dependents exist, and a rule cannot be inferred, the target is assumed to be "up to date".

Example:

```
.c.obj:
   icc -c -Fo$@  $<

hello.obj : hello.c
```

Similarly, given a target without a dependent or a command block, NMAKE32 will infer a dependent file from the targets base file name and all inference rules defined with the target's file extension. The inference rules are searched in the order that they are defined in the .SUFFIXES target.

Example:

```
.SUFFIXES: .c .asm .obj .exe

.obj.exe:
   icc -fe$@ $<

.c.obj:
   icc -c -f$@ $<

.asm.obj:
     masm $<;

target.exe: header.h
target.obj:
```

In the preceding example, NMAKE32 will search for an inference rule that builds a .exe file, it will locate the .exe.obj rule, using the target filename and the source extension, target.obj will be inferred as a dependent. It will then check to see if it can locate an up-to-date target.obj on disk, or can build one. When NMAKE32 locates target.obj as a target defined in the description file, it will try to build it. In order to build this file it will use the same inference rule process to infer the dependent target.c (searching extensions in the order given by .SUFFIXES). If target.c did not exist on disk or could be built from the description file, the next inference rule evaluated would be .asm.obj:, and the next inferred dependent would be target.asm.

Default Inference Rules

The use of inference rules eliminates the need to put the same commands in several description blocks. Default inference rules and extensions can be updated by the TOOLS.INI, BUILTINS.MAK, or the file specified by the /b option on the command line (see The TOOLS.INI File and The BUILTINS.MAK File).

Default inference rules and extensions are supplied by NMAKE32. The .SUFFIXES: target defaults to the following extensions:

```
   .SUFFIXES: .exe .obj .asm .c .bas .cbl .for .pas .res .rc .cpp .cxx
```

The default inference extensions can be eliminated by defining an empty .SUFFIXES: target. The .SUFFIXES: target can be redefined as needed.

Example:

```
.SUFFIXES:
.SUFFIXES:  .exe  .obj  .asm  .c
```

The list of default inference rules are the following:

```
.asm.exe:
    $(AS) $(AFLAGS) /c $<;

.asm.obj:
    $(AS) $(AFLAGS) $<;

.c.exe:
    $(CC) $(CFLAGS) $<

.c.obj:
    $(CC) $(CFLAGS) /c $<

.cbl.exe:
    $(COBOL) $(COBFLAGS) $<, $@;

.cbl.obj:
    $(COBOL) $(COBFLAGS) $<;

.for.exe:
    $(FOR) $(FFLAGS) $<

.for.obj:
    $(FOR) /c $(FFLAGS) $<

.pas.exe:
    $(PASCAL) $(PFLAGS) $<

.pas.obj:
    $(PASCAL) /c $(PFLAGS) $<

.rc.res:
    $(RC) $(RFLAGS) /r $<

.cpp.obj:
    $(CC) $(CFLAGS) /c $<

.cxx.obj:
    $(CC) $(CFLAGS) /c $<
```

Inference Rules with Curly Braces

Inference rules can be combined with specified target/dependency paths for each rule. Given the inference rule below, if a dependent file with the given extension is found in the dep_path directory during inference rule processing, then the target will be updated in the specified target_path directory. These paths are specified in the inference rules inside curly braces. Only single paths are supported inside curly braces on inference rules. However, multiple inference rules using the same extensions and different paths can be specified. The first matching inference rule is the rule that is used if multiple inference rules are specified in the description file.

```
{dep_path}.extension1{target_path}.extension2:
    command block
```

Example:

```
# Build objects in the p2 directory from source
# in the p1 directory

{p1}.c{p2}.obj:
    icc /c /Fo$@  $<

p2\file.obj: p1\file.c
```

When using paths or target/dependency statements and curly brace paths on inference rules, all paths must be carefully matched. When a target/dependency statement dependent path does not match the curly brace path on the inference dependent extension, it is deemed a mismatch. The following example combines inference rules, paths and explicit rules to demonstrate correct application of the rule concepts.

Example:

```
{p1}.c{p2}.obj:                 #  Valid .SUFFIXES: extensions
    $CC) $(CFLAGS) /Fo$@ $<     #  First inference rule

{p3}.c{p4}.obj:                 #  Valid .SUFFIXES: extensions
    $(CC) $(CPPFLAGS) /Fo$@ $<  #  Second inference rule

{p1}.c{p4}.obj:                 #  Valid .SUFFIXES: extensions
    $(CC) $(CPPFLAGS) /Fo$@ $<  #  Third inference rule

p2\dep.obj : p1\dep.c           #  Target / dependency statement
    @echo Bogus explicit rule   #  Explicit rule

p4\dep.obj : p1\dep.c           #  Target / dependency statement
                                #  with no explicit rule
```

The application of the inference rules for the target/dependency with no explicit rule is as follows:

1. The first rule does not apply, because it is an inference rule to create objects in the p2 directory from source in the p1 directory.
2. The second rule does not apply, because it is an inference rule to create objects in the p4 directory from source in the p3 directory.
3. The third inference rule creates objects in the p4 directory from source in the p1 directory. This rule is used.
4. Although the paths for the dependent and target on the first target\dependency statement match the first inference rule, the inclusion of an explicit rule in this dependency precludes it from being built using any inference rule processing.

The third inference rule is the rule that applies to the second target/dependency statement. The target will be updated in the p4 directory from the dependent in the p1 directory, if required.

Inference Rule and Curly Braces on Target/Dependents

When curly brace paths are used on target/dependency statements, the location of existing target/dependency files determines the inference rule to be used. This is demonstrated in the following example:

Example:

```
{p1}.c{p4}.obj:
   icc -fo$@ -c $<

{p3}.c{p4}.obj:
   icc -fo$@ -c $<
```

```
{p3}.c{p4}.obj:
   icc -fo$@ -c $<

{p2;p4} file.obj : {p1;p3} file.c
```

If file.obj exists in the p4 directory and file.c exists in the p1 directory, the first inference rule will be used if the target needs updating. Note that if file.obj also was on disk in the p2 directory, the first directory found during explicit target path matching would be p2. There is not an inference rule defined for the {p1}.c{p2}.obj combination, so inference rule processing would fail.

Built-in Commands

Built-in commands are operations handled internally by NMAKE32, rather than passed out to the operating system. All built-in commands begin with a percent sign (%).

%cd *directory*	Instructs NMAKE32 to change the current directory to *directory* .
%do *other_target*	Instructs NMAKE32 to invoke the commands of target *other_target* as though they were inline. All built-in macros (such as $@ and $?) associated with the current target and dependents remain unchanged; only the commands are used. This built-in command can be used to set up generic rules which can be invoked repetitively from different points in your description file. Note that it does not matter where *other_target* was defined, as long as it is visible to NMAKE32 during the processing of your description file.

Example:

```
  # Use the %do command in the target/dependency
  # command using the target comp_rule:  This target
  # is not listed as a dependent of any other target,
  # and is only accessible via the %do command.

  comp_rule:
     $(CC) -c -fo$@ $<

  all: test1.obj test2.obj        # define targets to build

  test1.obj: test1.c              # first target/dep pair
    %do comp_rule                 #   %do command

  test2.obj: test2.c              # second target/dep pair
    $(CC) /c -fo$@ $<             # use explicit command
```

%echo *string*	Instructs NMAKE32 to display the contents of string (after expanding any macros)
%set *var=value*	Instructs NMAKE32 to assign value to the variable *var* . There are instances where it would be a valuable feature to be able to change the value of a variable after NMAKE32 has read all the description files and has begun execution. The %set command allows to you to change or create any NMAKE32 variable while commands are being invoked.
%setenv *var=value*	Instructs NMAKE32 to assign value to the environment variable *var* . If the variable is not already defined, it is created with the specified value; if the variable is already defined, its value is replaced by the specified value.

Pseudotargets

A pseudotarget is a target in a description block that is not a file. It is a name that serves as a handle for building a group of files or executing a command block. There are two types of pseudotargets recognized by NMAKE32, predefined and user defined.

Predefined pseudotargets start with a period (.) and are used to control NMAKE32 processing characteristics. NMAKE32 provided pseudotargets and function follow:

.IGNORE: [targets]	Tells NMAKE32 to ignore return codes from invoked commands. Same function as /i option on the command line. If specified with targets, only return codes of those targets are ignored.
.MAKEINIT: [command]	The commands of this target are executed just before the first real target is examined (after the command line and the description file have been processed).
.INIT: [command]	The commands of this target are executed just before the first real target is built (just before the first user command is invoked).
.MAKEDEINIT: [command]	The commands of this target are executed just before NMAKE32 terminates.
.DEINIT: [command]	The commands of this target are executed just before the commands of the .MAKEDEINIT: target are executed, but only if the commands of the .INIT: target and at least one user command have been executed.
.RECHECK:	Instructs NMAKE32 to recheck the date, time and path of each target (the check occurs after all commands for that target have been executed).
.NORECHECK:	Instructs NMAKE32 to not recheck the date, time and path of each target. This is the default behavior.
.PRECIOUS: [targets]	Instructs NMAKE32 not to delete the targets if the command completes with a nonzero return code. If specified with targets, only those targets are saved for nonzero return codes.
.SILENT:	Instructs NMAKE32 not to echo commands. Same function as /s option on the command line.
.SUFFIXES: [exts]	Used to change the defined extensions that NMAKE32 recognizes for inference rule processing. If specified without extensions, resets the defined extensions to null. If followed by extensions, these are added to the defined recognized extensions.

Example:

```
# Example of .SUFFIXES and .PRECIOUS

.sav.exe:
    -copy keep.sav keepthis.exe

.SUFFIXES:                 # This set .SUFFIXES to null
.SUFFIXES: .sav .exe       # These two extensions can be used
                           # in inference rules

.PRECIOUS:   keepthis.exe  # This protects keepthis.exe from
                           # being deleted

keepthis.exe :keep.sav
                           #  Inference rule will be used here
```

User defined pseudotargets are used when you wish to invoke a block of commands without having them associated to a target. This would allow you to invoke the block of commands either from the command line by specifying the pseudotarget name, or from the description file via a dependency. (The commands for a pseudotarget without dependencies are always executed.)

Example:

```
ALWAYS:
  @echo  These commands will always execute if you
  @echo  specify the pseudotarget ALWAYS on the NMAKE32
  @echo  invocation, or you run a description file where
  @echo  ALWAYS appears as a dependent, or the only target
  @echo  in the description file.  There is no actual
  @echo  file named ALWAYS.
```

Command Line Modifiers

The NMAKE32 syntax includes prefixes that change the way it handles command processing. By prepending these special characters in front of a command you can alter NMAKE32 default behavior for a given command.

The following are the defined prefixes and their usages:

@*command*	Execute *command* in silent mode. Prevents the command from being displayed.
-*command*	Ignore the return codes from *command*. This prefix sets the %status built-in function with the return code of *command*, but prevents NMAKE32 execution interruption if it was non-zero. Similar function to the /i option on the command line.
-n *command*	Ignore all return codes for *command* less than or equal to n.
~*command*	Propagate the return code from the previously executed command and prevent NMAKE32 execution interruption if *command* returns a non-zero value. When used in conjunction with the %status() built-in function (for use in !if expressions) returns the value of the last command's return code. This combination is very useful for doing post-processing on commands, even if those commands have failed.

Example:

This example will execute a command that has a non-zero return code and NOT interrupt NMAKE32 execution. It will propagate the return code from that command for later processing.

```
#  Execute an intentionally bad command

target:
   -copy garbage.txt filename

!if %status()==1
   ~@echo the %status is set to 1
   ~@echo these commands would usually update %status
!endif

copy hello.c hello.z

!if %status()==0
  @echo  the status value is 0
```

!*command*	Execute the command for each dependent in the $? macro values.
=*command*	Force date, time and path rechecking for the current target after all commands for the target have been executed.
&*command*	Execute this command, even if the -N flag was specified.

Note: The built-in commands, such as %echo and %cd, ignore the command line modifiers, and the -n command line modifier cannot be used with a built-in command. Remember the built-in commands are handled internally, so they do not provide return codes and are not echoed.

Macros

Macros provide a convenient way to replace one string with another in the description file. The text is automatically replaced each time NMAKE32 is run. This feature makes it easy to change text throughout the description file without having to edit every line that uses the text. Some common uses of macros are:

- To create a standard description file for several projects. The macro represents the names of commands. These names are defined when you run NMAKE32. When a different project is started, changing the macro changes the names NMAKE32 uses throughout the description file.
- To control the options that NMAKE32 passes to the compiler, assembler, or linker. When using a macro to specify the options, options can be quickly changed throughout the description file in one easy step.
- Defining groups of files in a project, such as the .OBJ files required to build an executable or a library.

A macro can be defined:

- On the command line
- As a predefined macro
- Through inheritance from environment variables
- In the TOOLS.INI or BUILTINS.MAK (if defined)
- In a description file

Defining Macros

A macro definition follows this form:

```
name=character-string
name+=character-string
name=+character-string
```

Macro names are case_sensitive by default. The *character-string* can be any string of characters. The first character of the macro must be in column one. NMAKE32 ignores spaces surrounding the *name* .

The *character-string* can be a null string and can contain embedded spaces. Do not enclose the macro string in quotation marks. Quotation marks are used when macros defined on the command line contain embedded spaces.

The "+=" form of a macro assignment causes the *character-string* value to be appended to the end of the current value of *name* , while the "=+" form causes *character-string* to be prepended to the beginning of the current value of *name* . Values are concatenated, spaces are not added.

NMAKE32 also allows the user to recursively define a macro, by using the macro name in the definition. This results in a permanent change to the original macro.

Examples:

Suppose you have a macro named CFLAGS which is used to define the flags for your C compiler, and you wish to add a flag. The original definition would be:

```
    CFLAGS = -Fo$@ -c
```

You can then append to this definition a new flag:

```
    CFLAGS += -Zi
```

The resulting string definition for CFLAGS is then:

```
-Fo$@ -c -Zi
```

If the new flag had been prepended:

```
CFLAGS =+ -Zi
```

The resulting string definition for CFLAGS is then:

```
-Zi -Fo$@ -c
```

This could also be done using recursion and appending a new flag:

```
CFLAGS = $(CFLAGS) -Zi
```

The resulting string definition for CFLAGS is:

```
-Fo$@ -c -Zi
```

Macros on the Command Line

A macro may also be defined on the command line when calling NMAKE32. Command-line macro definitions follow this form:

name=*character-string*

If the macro contains embedded spaces or special characters defined in the shell, enclose it in double quotation marks (").

Inherited Macros

NMAKE32 inherits all current environment variables as macros. For example, if you have a PATH environment variable defined as PATH = C:\TOOLS\BIN, the string C:\TOOLS\BIN is substituted when you use $(PATH) in the description file.

Inherited macros may be redefined by including a line such as the one in the example above in a description file. While NMAKE32 is executing, the macro takes on the redefined definition. However, when NMAKE32 terminates the environment variable resumes its original value.

The /E (override environment variable) option disables inherited macro redefinition. If you use this option, NMAKE32 ignores any attempt to redefine an inherited macro, except from the command line.

Referencing Macros

After a macro has been defined, it can be used anywhere in a description file using the following syntax:

```
$(name)
```

The parentheses are not required if the macro name is only one character long. To use a dollar sign ($) without using a macro, enter two dollar signs ($$), or use the caret (^) before the dollar sign as an escape character.

When NMAKE32 runs, it replaces all occurrences of $(*name*) with the defined macro string. If the macro is undefined, a null string is substituted. Defined macros can be undefined with the !UNDEF directive.

Example:

To echo the following line to the screen:

```
$(HELLO) is HI
```

The macro must be referenced with double dollar signs ($$), as shown in the following description file excerpt:

```
HELLO = HI

dummy:
     @echo $$(HELLO) is $(HELLO)
```

Macro Substitutions

NMAKE32 allows substitutions in macro invocations that allow a user to change the value generated without changing the macro itself. The following syntax is used for macro substitutions:

`$(`*`name`*`: `*`character-string1`*` = `*`character-string2`*`)`

where *name* is the name of the macro whose value is being modified, *character-string1* is character or characters between the colon ":" and the equal "=" sign, and *character-string2* is the replacement character or characters after the equal "=" sign and before the closing parentheses ")". If *character-string2* is null, *.character-string1* is removed from *name* (that is, replaced by a null string).

Example:

The following macro:

```
FILES = file1.z file2.z file3.z
```

followed by this macro substitution:

```
$(FILES:.z=.c)
```

expands to the value:

```
file1.c file2.c file3.c
```

The actual value of FILES remains unchanged.

Built-in Macros

File-Specification Parts

A full file specification gives the base name of the file, the file name extension, and the path. The path provides the disk drive identifier and the sequence of directories needed to locate the file on the disk.

Example:

The file specification:

```
C:\SOURCE\PROG\SORT.OBJ
```

has the following parts:

Path Name	C:\SOURCE\PROG	everything prior to the last path

		separator
File Extension	.OBJ	everything after and including the last dot (.)
Base File Name	SORT	what's left

NMAKE32 predefines several macros which are useful for writing rules for building targets.

Macro	Value
$$	The '$' character, which does not participate in any further macro expansion.
$*	The path name and base file name of the current target with the extension deleted.
$<	When used in an explicit rule; the first dependent of the target (regardless of whether the dependent is out of date), if any. When used in an inference rule, the file specification of the (inferred) dependent which is out of date with respect to the target. For example, in the .c.obj: rule, $< would evaluate to the .c file.
$@	The file specification of the current target.
$?	The file specification of all dependents which are out of date with respect to the target.
$**	The list of file specifications of all dependents (whether up-to-date or not).
$$@	The file specification of the target that NMAKE32 is currently evaluating. This is a dynamic dependency parameter, used only in dependent lists.
$:	The path name of the current target file specification.
$.	The base file name and extension of the current target file specification.
$&	The base file name of the current target file specification.
$()	The null string

Note: The special macros $** and $$@ are the only exceptions to the rule that macro names longer than one character must be enclosed in parentheses.

Example:

Given the following description file excerpt:

```
     c:\mydir\myprog.obj: $*.c
          @echo Dollar Star is $*
          @echo Dollar Star Star is $**
          @echo Dollar At is $@
```

The special macro $* on the target dependent line will expand to `c:\mydir\myprog`, and the following text will be displayed when NMAKE32 is run:

```
     Dollar Star is c:\mydir\myprog
     Dollar Star Star is c:\mydir\myprog.c
     Dollar At is c:\mydir\myprog.obj
```

Characters that Modify Built-in Macros

The following macros all resolve to a file specification (or possible several file specifications for $** and $?):

```
$*     $@     $**     $<     $?     $$@
```

There are four characters (D, F, B, or R), which may be appended to any of these macros to modify the filename returned by the macro. Parts of the full file specification are returned, depending on which character is used.

	Appended Character			
File Part Returned	D	F	B	R
File Path	Yes	No	No	Yes
Base File Name	No	Yes	Yes	Yes
File Name Extension	No	Yes	No	No

Example:

If the macro $@ has the value

```
C:\SOURCE\PROG\SORT.OBJ
```

then the following values are returned for the modified macro:

Macro	Value
$(@D)	C:\SOURCE\PROG
$(@F)	SORT.OBJ
$(@B)	SORT
$(@R)	C:\SOURCE\PROG\SORT

Note: Modified macros are always longer than a single character so they must be enclosed by parentheses when used.

Predefined Macros

The following macros are defined (or read from the environment variable) by NMAKE32 before the first file is read.

MAKEDIR	Current working directory when NMAKE32 is invoked.
MAKE	The name of the NMAKE32 program.
MAKEVER	The current NMAKE32 version number.
MAKEFLAGS	All command line parameters, plus the values form the MAKEFLAGS environment variable, except for b, f, o, p, and x options.
TIMESTAMP	Data and time when NMAKE32 was started. The format is locale specific.
DATE	Date from TIMESTAMP in the form: YYYYMMDD (for example, 19960325)
TIME	Time from TIMESTAMP in the form: HHMMSS (for example, 123015)
AS	Expands to the default assembler, which is "masm" on OS/2.

CC	Expands to the default C language compiler, which is "icc" on OS/2.
COBOL	Expands to the default cobol compiler, which is "cobol" on OS/2.
FOR	Expands to the default fortran compiler, which is "fl" on OS/2.
PASCAL	Expands to the default pascal compiler, which is "pl" on OS/2.
RC	Expands to "rc", the resource compiler, which is "rc" on OS/2.

The following macro, while not predefined by NMAKE32, also has special significance

.PATH.*ext* = *path_list*

where:

ext	is the extension for which the search path should be applied.
path_list	is the list of paths, separated by semicolons. Note that macros are expanded in this path list, so $(INCLUDE), $(DPATH), or any other macro or environment variable could be used.

The *path_list* is only used when a dependent file specification has no path and cannot be found as specified. NMAKE32 tries to locate the file in each of the paths specified in the *path_list* which matches that file's extension. It uses the first copy of the file found in the path list.

Any number of these macros (one per extension) may be defined in a description file, so that any or all of the file extensions that are specified in your description file may be supported.

Extmake Syntax

Description files can use a special syntax to determine the drive, path, base name, and extension of the first dependent file in a description block. This syntax is called the "extmake" syntax.

The characters %s represent the complete file specification of the first dependent file. Various parts of the file specification are represented using the syntax:

```
%|<parts>F
```

where *<parts>* is any combination of the following letters:

d	Drive
p	Path
f	Base name
e	Extension

The percent symbol (%) is a replacement in the OS/2 command line. To use the percent symbol in the command-line arguments, use a double percent (%%).

Example:

Using the following description file excerpt:

```
d:\path\filename.ext
@echo"%%s" is "%s" and "%%|dpfeF" is "%|dpfeF"
```

The following would be displayed:

```
"%s" is "d:\path\filename.ext" and "%|dpfeF" is
"d:\path\filename.ext"
```

In this example a double percent sign is used to display a single percent sign, and the output from %s is that same as the output from %|dpfeF.

Transformation Macros

Transformation macros are used to manipulate strings, which in turn can contain macros. Macros are evaluated from the "inside out" so that the strings will be fully resolved before the transformation is performed on them. Note: When using the transformation macros described below, there must be no spaces between the initial dollar sign and the first comma, or the macro will not be recognized.

NMAKE32 supports the following transformation macros:

Include Transformation

Syntax:

```
$[@,filespec]
```

In this transformation, NMAKE32 will return the contents of the specified file, as a single string with comments removed.

Example:

INCLUDE.FIL is a file with the following lines:

```
# This is a comment and will not be returned
Line one,
            # This comment will not be returned
 and line two.
```

The transformation macro:

```
$[@,INCLUDE.FIL]
```

returns:

```
Line one, and line two.
```

Clip String Transformation

Syntax:

```
$[c,string,start,end]
```

In this transformation, NMAKE32 will return characters at positions *start* through *end*, inclusive, where *start* and *end* are integers. If *start* is not specified, it defaults to 1. If *end* is not specified, it defaults to the end of the string. Note that the first position is 1, not 0. If either *start* or *end* is a negative number, it indicates the position relative to the end of the string, rather than to the start. In this case, -1 indicates the last character.

Examples:

Given the following clip string transformations:

```
$[c,FILE01.OBJ,5,6]
$[c,FILE01.OBJ,-6]
$[c,FILE01.OBJ,,6]
```

The first transformation will return:

```
01
```

The second transformation will return:

```
01.OBJ
```

The third transformation will return:

```
FILE01
```

Directory Transformation

Syntax:

```
$[d,string]
```

In this transformation, NMAKE32 will return the subdirectory portion of *string* . If there is no drive or path specified in *string* , this transformation will return an empty string; otherwise, it may contain a drive specifier, a path specification, or both.

Example:

The directory transformation:

```
$[d,D:\MAKE\TEST.FIL]
```

returns:

```
D:\MAKE
```

Note: Unlike the "path" transformation, the directory transformation will never end with a path separator.

Extension Transformation

Syntax:

```
$[e,string]
```

In this transformation, NMAKE32 will return the file extension portion of *string* . If there is no file extension specified in *string* , this transformation will return an empty string.

Example:

The extension transformation:

```
     $[e,D:\MAKE\TEST.FIL]
```

returns:

```
     FIL
```

Filename Transformation

Syntax:

```
     $[f,path,rootlist,extension]
```

In this transformation, NMAKE32 uses the parts specified to construct complete file specification. The *rootlist* value may contain one or more filenames, each of which can be fully-qualified. NMAKE32 takes each of those names, adds the value specified by *path* and the value specified by *extension* (replacing those parts if specified in the name), and returns a string containing these new file specifications. If the *path* or *extension* value is omitted, that part of the original file specification will remain unchanged; however, if either of those values is a null string (""), the corresponding parts of the file specification will be removed.

Examples:

The transformation macros:

```
     $[f,C:\OS2\,D:\MAKE\TEST.FIL,EXT]
     $[f,"",D:\MAKE\TEST.FIL,]
```

returns:

```
     C:\OS2\TEST.EXT
     MAKE.FIL
```

Lowercase Transformation

Syntax:

```
     $[l,string]
```

In this transformation, NMAKE32 will return the lowercase equivalent of *string* , using the current locale.

Example:

The following lowercase transformation:

```
     $[l,BIG]
```

returns:

```
big
```

Filename Pattern Matching Transformation

Syntax:

```
$[m,pattern,string]
```

In this transformation, NMAKE32 will return those words in string which match the filename pattern specified by pattern. The matching is **case-insensitive**.

A filename can contain one or more of the following patterns:

*	Matches any string, including the null string.
?	Matches any single character.
[...]	Matches any one of the enclosed characters.
[.-.]	Matches any character between the enclosed pair, inclusive (range)

To remove the special meaning of the characters {, }, \, ., *, ^, and !, if they are part of the filename to be matched, precede them with a backslash. If the dollar ($) character is used in a pattern, it must be preceded by a caret (^) character so that it will not be interpreted as a macro. The characters [,], and comma (,) cannot be used in pattern matching.

Enclosed characters can be combined with ranges. Therefore, [ABCM-Z]* matches any filename that begins with A, B, C, or M through Z.

Examples:

The following transformation:

```
$[m,*.c,x.c a.h b.h c.h]
```

returns:

```
x.c
```

Since *string* can include macros as well as text, a transformation macro can be used to easily extract dependents with a particular extension. If the following line was in a description file:

```
my.obj: my.c a.h b.h c.h
```

The macro:

```
$[m,*.c,$**]
```

returns:

```
my.c
```

Regular Expression Matching Transformation

Syntax:

```
$[mr,regexp,string]
```

In this transformation, NMAKE32 will return those words in *string* which match the regular expression specified by *regexp* . The matching is case-insensitive.

The following expressions match a single character:

c	Any ordinary character, other than one of the special pattern-matching characters, matches itself.
.	A period (.) matches any single character.
[string]	A string enclosed in square brackets matches any one character in the string
[.-.]	A range is two characters separated by a dash and enclosed in square brackets. It matches any character that is within the range.
[ˆstring]	A string (or range) enclosed in square brackets and preceded by a caret (ˆ) matches any character except for the character in the string (or range). Strings and ranges may be combined as needed, as in: [a-m0-9xyz], which matches a thru m, 0 thru 9, x, y, or, z.
\c	The backslash (\) character followed by any character matches that character. This is useful for matching the following special characters; . *{ } ˆ \ The dollar ($) sign must be preceded by a caret (ˆ) character so it will not be interpreted as a macro. The characters [,], and comma (,) cannot be used in pattern matching.

The single-character expressions can be combined into regular expressions as follows:

*	Match zero or more occurrences of the previous character.

A regular expression can be restricted to match text that begins on the first character of the string, ends on the last character of the string, or both, as follows:

^pattern	The pattern matches text that begins on the first character of the string.

Example:

This example is the same as the filename pattern matching example, but the pattern is written as a regular expression:

```
$[mr,.*\.c,x.c a.h b.h c.h]
```

returns:

```
x.c
```

Path Transformation

Syntax:

```
$[p,string]
```

In this transformation, NMAKE32 will return the path portion of *string*. If there is no drive or path specified in *string*, this transformation will return an empty string; otherwise, it may contain a drive specifier, a path specification, or both.

Example:

The following path transformation:

```
$[p,D:\MAKE\TEST.FIL]
```

returns:

```
D:\MAKE\
```

Note: Unlike the directory transformation, the path transformation always ends with a path separator (if path information is present in string)

Root Transformation

Syntax:

```
$[r,string]
```

In this transformation, NMAKE32 will return the base file name portion of *string*. If there is no file name specified in *string*, this transformation will return an empty string.

Example:

The following root transformation:

```
$[r,D:\MAKE\TEST.FIL]
```

returns:

```
TEST
```

Separator Transformation

Syntax:

```
$[s,separator,string]
```

In this transformation, NMAKE32 will place the separator text between every two words in *string*. This transformation is useful when a character other than a space is required to separate multiple words on a program's command lines, or when writing in-line files.

The *separator* text can be enclosed in quotes, and if so, can contain special characters such as spaces, commas, and the following escape sequences:

Escape Sequence	Description
\"	Double quote
\n	New line
\r	Carriage return
\t	Tab
\\	Backslash
\nnn	Character with octal value nnn
\xnn	Character with hex value xnn

Example:

When creating a response file for a program, such as LINK386 which has a limit to the length of any single line in the response file, and a method for continuing that single logical line across several physical lines, the following transformation macro should be used. Assuming the line continuation character is a plus sign(+), and there is a long list of object modules in the macro OBJS:

```
OBJS = p1.obj p2.obj p3.obj
```

The transformation:

```
$[s,"+\n",$(OBJS)]
```

would return

```
p1.obj+
p2.obj+
p3.obj
```

Using this macro, the line limit will not be exceeded, and they can be put together in one logical line.

In the above example, NMAKE32 will take a list of object file specifications and place one per line in a response file. However, since NMAKE32 never changes the text, the file specifications will end up as they were specified, without information such as paths.

Since the expansion of the built-in macros can yield path information, the separate and pattern matching transformations can be nested for more flexibility.

If the following line was in a description file:

```
prog.exe: $(OBJS) prof.def other.lib
```

The following transformation:

```
$[s,"+\n",$[m,*.obj,$**]]
```

would return only those dependents of prog.exe that are object modules, but with any path information that NMAKE32 was able to determine, and write them one per line.

```
<path for p1>\p1.obj+
<path for p2>\p2.obj+
<path for p3>\p3.obj
```

Translation Transformation

Syntax:

```
$[t,*input*,*output*,*string*]
```

In this transformation, NMAKE32 will examine the contents of *string* and translate all occurrences of the characters in *input* to the corresponding characters in *output*.

If *output* is null, all characters in *input* are removed from *string*. If *output* is shorter than *input*, the characters of *output* are reused.

Examples:

The following transformations:

```
$[t,/\,\/,d:\make\makemac.c]
$[t,13579,*,1234567890]
$[t,13579,,1234567890]
```

return

```
d:/make/makemac.c
*2*4*6*8*0
24680
```

The *input* and *output* strings can be enclosed in quotes and, if so, can contain special characters such as spaces, commas, and escape sequences as in the separate transformations ($[s,...]).

Uppercase Transformation

Syntax:

```
$[u,*string*]
```

In this transformation, NMAKE32 will return the uppercase equivalent of *string*, based on the current locale.

Example:

The following transformation:

```
$[u,upper]
```

returns:

```
UPPER
```

Macro Expansion and Nesting

To fully understand the effects of macros in a description file it is useful to know a little about how NMAKE32 operates. During read-time, NMAKE32 expands all macros except those occurring within a command block. Macros used in command blocks are expanded at run-time, and they will have the last value defined at read-time.

Example:

Given the following description file excerpt:

```
MAC = 1
!IF "$(MAC)" == "1"
target0:
   @ECHO The value was 1
   @ECHO $$(MAC) is $(MAC)
!ENDIF
MAC = 2
```

returns:

```
The value was 1
$(MAC) is 2
```

The !IF directive will evaluate to non-zero and execute the command block for target0 because directives are processed at read-time, and at this time the value of MAC is 1. Since MAC is redefined at the end of the description file, the run-time value is now changed to 2. The target0 command block is executed at run-time so the value displayed for the definition of MAC is 2.

Macro references may also be nested. Macros are said to be nested if an expanded macro is used as part of another macro reference.

Example:

If the following macros are defined:

```
A = X
B = Y
C = Z
$XYZ = hello
```

then a reference to the following macro:

```
$($A$B$C)
```

would have the definition `hello`.

Directives

NMAKE32 provides directives that:

- Conditionally execute commands
- Display error messages
- Include the contents of other files
- Turn some NMAKE32 options on or off

Each of these directives controls description file processing, rather than target processing. Read-time directives should begin with an exclamation point (!) and must start in column one. Run-time directives must be preceded by one or more spaces and must start with a percent sign (%). Spaces may exist between the conditional character and the rest of the line. For example, "! if *expression* " is the same as "!if *expression* ".

The list below describes the directives:

Note: This table shows directives defined using an exclamation point.

Directive syntax	Usage
!if *expression*	Processes the statements

	between the !if keyword and the next !else, !elif, or !endif directive if *expression* evaluates to a non-zero value (TRUE). Otherwise, the lines are ignored.
!else	Processes the statements between the !else and the !elif or !endif directive if the preceeding !if, !elif, !ifdef, or !ifndef expression evaluated to zero (FALSE).
!elif *expression*	Is identical to !else processing except that the lines which follow are processed only if the new expression evaluates to a non-zero value.
!elseif *expression*	Is a synonym for !elif *expression*.
!endif	Marks the end of a !if, !ifdef, or !ifndef block of statements.
!foreach var [in] word_list	Processes all statements up through the ending !endfor once for each word in word_list. During each iteration the variable var will be set to the corresponding word from the list. This macro can be referenced as any other macro would be referenced, via $(var). After all iterations are complete, the macro will retain the last value.
!endfor	Marks the end of a !foreach block of statements.
!ifdef *macro*	Is identical to !if processing, except that the statements which follow are processed only if the *macro* variable is currently defined. Note that variables defined to be the null string are still considered to be defined for !ifdef processing. For compatibility with previous versions of MAKE, the form !ifdef $(macro) is still supported.
!ifndef *macro*	Is identical to !ifdef processing, except that the lines which follow are processed only if the macro variable is not currently defined. Note that variables defined to be the null string are still considered to be defined for !ifndef processing.
!undef *macro*	Undefines a previously defined macro. Subsequent references to $(macro) will return an empty string, and !ifdef *macro* will evaluate to zero.
!error *text*	Prints out *text* (macros are expanded) and then immediately

	terminates NMAKE32 with a return code of 1.
!include [<*filename*>]	Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (<>), NMAKE32 searches for the file in the directories specified by the INCLUDE macro; otherwise, it looks only in the directory specified. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.
!tryinclude [<*filename*>]	Similar to !include, except if the file does not exist, NMAKE32 will ignore the error and continue.
!cmdswitches {+|-}<opt>	Turns on or off one of four NMAKE32 options: /D, /I, /N, and /S. If no options are specified, the options are reset to the values they had when NMAKE32 was started. To turn an option on, precede it with a plus sign (+); to turn it off precede it with a minus sign (-). This directive updates the MAKEFLAGS macro.

Expressions Supported by NMAKE32

The expression used with the !if, !elif, and !elseif directives can consist of integer constants, string constants, built-in functions, or exit codes returned by programs. Integer constants can use the C unary operators for numerical negation (-), one's complement (˜), and logical negation (!). Any of the C binary operators listed below any also be used:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
ˆ	Exponentiation
&&	Logical AND
||	Logical OR
==	Equality
!=	Inequality
<	Less than
>	Greater than

<=	Less than or equal to
>=	Greater than or equal to

Directive Notes

- Directives can be nested to any depth, limited only by available memory.
- Included files can include other files as well, loops are not detected.
- Parentheses must be used to group expressions if a directive contains multiple expressions.
- Values are assumed to be decimal values unless specified with a leading 0 (octal) or a leading 0x (hexadecimal).
- Strings are enclosed by double quotes ("). You can use the equality (==) and inequality (!=) operators to compare two strings.
- A program can be invoked in an expression by enclosing the program name in square brackets ([ ]). The exit code returned by the program is used in the expression. Such programs will always be invoked if the directive is evaluated, even if -N has been specified.

Example

Using the following description file excerpt:

```
!INCLUDE <INFRULES.TXT>
!CMDSWITCHES +D
WINNER.EXE: WINNER.OBJ
!IFDEF DEBUG
!   IF "$(DEBUG)" == "y"
       ilink /DE WINNER.OBJ;
!   ELSE
       ilink WINNER.OBJ
!   ENDIF
!ELSE
!    ERROR Macro named DEBUG is not defined.
!ENDIF
```

The !INCLUDE directive causes the file INFRULES.TXT to be read and evaluated as if it were part of the description file.

The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked.

The !IFDEF directive checks to see whether the macro DEBUG is defined. If it is defined, the !IF directive checks to see whether it is set to 'y'. If it is, the linker is invoked with the /DE option; otherwise, it is invoked without the /DE. If the DEBUG macro is not defined, the !ERROR directive prints out the message and NMAKE32 stops executing.

Built-in Functions

Built-in functions test certain conditions at either read-time or run-time. They are used in expressions with the conditional constructs. The following built-in functions are available:

```
Function Name                Description

%defined(macro)              Returns 1 if macro is defined;
                             0 otherwise

%dir(dirspec)                Returns 1 if dirspec specifies
```

	a valid and existing directory; 0 otherwise
`%exist(`*`filespec`*`)`	Returns 1 if *filespec* exists (as a file or directory); 0 otherwise
`%exists(`*`filespec`*`)`	Is a synonym for %exist(*filespec*)
`%file(`*`filespec`*`)`	Returns 1 if *filespec* specifies a valid and existing file; 0 otherwise
`%member(`*`word`*`, `*`word_list`*`)`	Returns 1 if *word* is contained -- as a separate, blank delimited word -- in *word_list* ; 0 otherwise. The search is case-insensitive.
`%status()`	Returns the return code of the last user command invoked.
`%writable(`*`filespec`*`)`	Returns 1 if *filespec* specifies a valid and existing directory or file that is not read-only; 0 otherwise.

Example:

Built-in functions can be used just like other values in expressions:

```
!if %defined(macro1) && ("$(macro2)" == "ON") &&
    %exists($(target))
```

This directive is evaluated to non-zero if macro1 is defined, macro2 has the definition "ON", and the file name defined in the macro target exists as a file or directory.

Other Commands used in !IF Expressions

The operating system commands SET, CD, and <drive>: can also be used in !IF expressions. Specifying these commands within !IF statements will guarantee that they are executed before any compile is processed at read-time.

The syntax for this type of !if expression is:

```
!if [command]
```

where the commands set, cd, and <drive>: are enclosed in square brackets.

Example:

The following line in a description file:

```
!if [cd \] == 0
```

will change the directory to the root directory and execute the statement(s) after the !if, if the return code from the command is 0.

In-Line Files

Occasionally, the commands given in a description file exceed the command-line limit of the operating system. To avoid this, NMAKE32 allows the user to generate in-line files which can be read as response files by other programs.

The syntax to generate an in-line file is:

```
target: dependents
   command  <<[filename]

<any text>
<any text>
.
.
.
<<[KEEP | NOKEEP]
```

where <any text> can be text, macros, file names, predefined macros, or anything valid for the command executing.

Note: Loops are not detected.

All of the text between the two sets of double less than signs (<<) is placed into an in-line file and given the name *filename* . The inline file can be referred to later by using *filename* , providing the keep option is specified. If *filename* is not given, NMAKE32 gives the file a unique name in the directory specified by the TMP environment variable. This temporary file is erased after NMAKE32 has processed the command block. The in-line file can be temporary or permanent. If you do not specify otherwise, or if you specify the keyword NOKEEP, the in-line is temporary. Specify KEEP to retain the file. If the -N flag was specified, NMAKE32 will display the contents of the in-line file.

Note: Blank lines and comments are not ignored if they occur in an in-line file.

Example:

Below is a description file excerpt which shows how to create an in-line file for the link program:

```
target.exe: file1.obj file2.obj file3.obj file4.obj lib1.lib lib2.lib
    ilink @<<
 $[s,"+\n",$[m,*.obj,$**]]
 $@
 $*.map
 $[s,"+",$[m,*.lib,$**]]
 ;
 <<
```

NMAKE32 creates a file, in this case the file name is determined at run-time, and places the following lines into it.

```
file1.obj+
file2.obj+
file3.obj+
file4.obj+
target.exe
target.map
lib1+lib2
;
```

NMAKE32 executes the command `ilink` with a response file using the name determined by NMAKE32, and erases the file.

Escape Characters

Several control characters are used by NMAKE32 in its syntax. These characters are:

```
(          )     #     $     ^     \
{          }     !     @     -     [newline]
```

To use one of these characters in a command and not have it interpreted by NMAKE32, a caret (^) is used in front of the character.

Examples:

The string:

```
BIG^#.C
```

is treated as

```
BIG#.C
```

With the caret, you can include a literal newline character in a description file. This capability is useful in macro definitions, as in the following example:

```
XYZ=abc^<enter>
def
```

where <enter> is a newline character.

The effect is equivalent to the effect of assigning the C-style string abc\ndef to the XYZ macro. Note that this effect differs from the effect of using the backslash (\) to continue a line. A newline character that follows a backslash is replaced with a space, unless the -\ option is specified. NMAKE32 ignores a caret that is not followed by any of the characters it sees in its syntax.

Note: Control characters in the makefile that are used in filenames, pathnames, and so on, that are not to be evaluated by NMAKE32 must be preceded by the caret (^) character. For example, to define a macro which contains the root directory specifier **C:\**, the backslash character must be preceded by a caret character, since the backslash character is used to concatenate lines in NMAKE32. The proper way to define this macro is:

```
root=C:^\
```

If the macro was coded without the caret character, such as in:

```
root=C:\
!ifdef root
    .
    .
    .
```

an error would result, since NMAKE32 would concatenate the root macro definition with the next line in the makefile. Placing a caret character before the backslash character prevents NMAKE32 from processing the backslash as a control character.

Notices

May 1999

Copyright Notices

Disclaimers

Trademarks
